

RTC: a distributed realtime control system toolkit

Thomas G. Lockhart*

Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT

The Jet Propulsion Laboratory (JPL) has built several optical interferometers using a common software framework developed for this purpose. The heart of this framework is the Realtime Control (RTC) software product. RTC has evolved from its initial implementation to include a powerful dynamic configuration capability and to use Common Object Request Broker Architecture (CORBA) technology for commanding and telemetry. This paper describes the current implementation of this toolkit.

Keywords: RTC, RICST, real-time, realtime, distributed computing, CORBA, JPL, SIM, Keck

1. INTRODUCTION

Optical interferometry has become a viable technique for astronomical observations with the convergence of high-speed general-purpose processors, high-bandwidth optical sensors and mechanical actuators, and distributed computing frameworks. JPL's Interferometry Realtime Systems and Software Group has developed a software toolkit to support our work in interferometry. The RTC product allows flexible command and control of software-based realtime servo systems at servo rates exceeding 10kHz, leading to effective bandwidths in excess of 1kHz. An application for the Keck Interferometer is described in [4848-12]. Early versions of this toolkit have been described elsewhere^{1,2,3} in conjunction with the Real-time Interferometer Control System Testbed (RICST), which was created to enable development of this software.

Building distributed applications exhibiting real-time performance requires integrating a wide range of capabilities. Many design choices must be made, balancing performance, development effort, maintenance effort, and other concerns to successfully deploy these complex systems. This toolkit supports these systems with:

1. A set of software components designed to operate in highly configurable real-time systems.
2. A set of non-real-time support components to configure and run the systems.
3. A design allowing inheritance of and extension to components, to facility adaptation into new environments.

The RTC software package was developed and has evolved to meet the needs of optical interferometer systems. These systems typically have the following attributes:

1. High rate servo loops controlling hardware.
2. High volume engineering and science telemetry.
3. Low-rate, time-sensitive status telemetry.
4. Distributed processing with multiple CPUs sharing an interface backplane, and with multiple CPUs across racks.
5. Real-time operating system (e.g. VxWorks) for hard real-time control loops.
6. Non-real-time operating system (e.g. Unix, Linux) for non-real-time components such as telemetry logging and user interfaces.
7. A mix of processor instruction sets, with heterogeneous endian conventions.
8. A mix of implementation languages, with C++ for real-time code and dedicated high-performance servers, and Java, Tcl, Python, Perl, and C++ for non-real-time clients and servers.

A full system will have the components shown in Figure 1. They include a configuration server, which provides information to components as they start up, and occasionally when they are reconfigured at runtime. Another component

* Thomas.Lockhart@jpl.nasa.gov; phone 1-818-354-7797; fax 1-818-393-4357; Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena CA 91109-8099

is a telemetry server, which manages the telemetry coming from various publishers in the system (but primarily from real-time components). The Configurator allows a user to alter the configuration managed by the configuration server, the GUI allows a user to command real-time components, the sequencer also commands components much like a user could do through the GUI. And finally, an archiver may be reading telemetry provided by all of the other components, writing the streams to disk for later analysis.

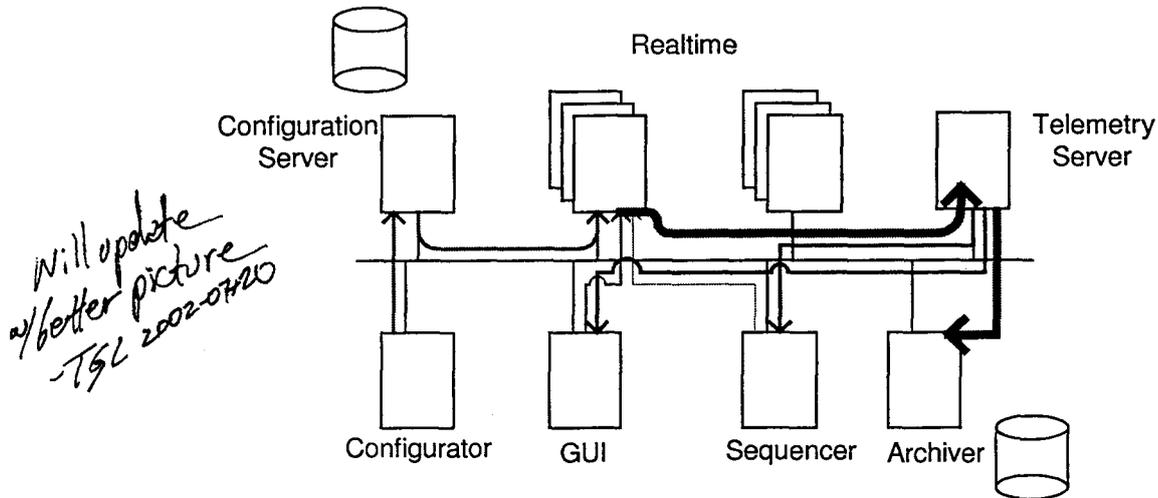


Figure 1. RTC Distributed Components

The software has evolved substantially since first developed in RICST. The most noticeable new features include a highly-capable object configuration infrastructure and the use of CORBA for our distributed communication. But in the process we have also more clearly separated interferometer-specific features from the generic capabilities required for any real-time system. This enhances our ability to bring basic features to a new system, allowing us to focus more quickly on developing unique new capabilities which may be required.

2. OBJECT-ORIENTED DESIGN

RTC has adopted object-oriented (OO) methodology as the most suitable approach for building scalable systems. Traditional procedural languages such as C and FORTRAN allow one to define functions or procedures, and to define data structures on which those procedures may act. The single most important feature of object-oriented languages is the concept of grouping functions and procedures together with data into an object, and having the attributes (data) and methods (functions) organized into a single entity. There are several defining characteristics for object-oriented software languages:

1. Objects. Data and code travel together. Classes are instantiated as objects having methods and attributes.
2. Inheritance. Objects can be extended via inheritance. Base objects implement what they need; children implement specializations or alternate behaviors.
3. Composition. Objects can own other objects.
4. Encapsulation. Objects hide information unless that information needs to be exposed outside the object.

Two attributes of modern object-oriented software, inheritability and extensibility, encourage a different approach to the design of applications using RTC, as compared with traditional procedural toolkits. In particular, one will find that for object-oriented toolkits component functionality and behaviors blend into the application layers above. In traditional procedural approaches, there tends to be a sharp line between a toolkit and an application. Understanding that there is a difference contributes to an easier path for design and implementation of applications using object-oriented techniques.

It may be noted that the earlier RICST development tended to stay away from a few C++ features such as virtual methods¹. Our experience has since shown that features such as virtual methods (which enable reusing implementation code from a base class for an inheriting child class) provide benefits in code compactness, conciseness, and clarity which outweigh potential performance degradation; which is measurably small and decreasing in significance with each new generation of faster processor.

3. REAL-TIME COMPONENT ARCHITECTURE

RTC provides systems with components to allow closed-loop control of complex hardware. These components include:

1. Device drivers to interact with hardware while insulating higher-level software from details about the device.
2. Sensors to read and interpret inputs.
3. Actuators to translate and write outputs.
4. Servos to control devices, using sensors for inputs to a calculation and using actuators to force a change based on the result of the calculation.
5. Periodic tasks to run servos. These tasks are run in a real-time environment which guarantees that the task will complete in its allotted time period, or report that the task did not complete within the specified interval.
6. Controller sets to manage sets of servos, allowing coordination of multi-stage servos.
7. High level objects (gizmos) which can accept commands and which manage periodic tasks containing servos.

We show these components as layers in a pyramid in Figure 2, with the high-level commandable gizmo at the top and the low-level device drivers at the bottom.

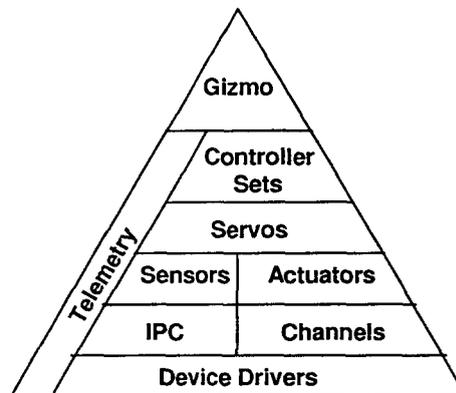


Figure 2. RTC Real-time Components

Components in each layer typically conform to an application programming interface (API) which exposes only the essential features of that component necessary for use by the layers above. For example, our device drivers typically implement Input and/or Output methods. One may correctly deduce that these APIs do not contain configuration information. This configuration information may vary from one component to another, and our configuration API enables hiding these implementation details; this API is discussed in a subsequent section.

A set of services is provided to support the RTC components mentioned above. These include:

1. Prioritized periodic task scheduling.
2. Precise timing.
3. Object commanding.
4. High-rate, high-volume telemetry handling.

Periodic task scheduling and precise timing is required for running reliable high-bandwidth control loops. RTC currently uses VxWorks for our production installations to provide real-time performance, though we use our own task scheduler to minimize the overhead and latency of scheduling and executing periodic tasks. We expect to see 20-80usec timing jitter from our production machines, though of course would prefer having less jitter for our highest-rate control loops.

Most high-level object commanding is done through CORBA-defined interfaces. Typically commands issued at this level are not time critical. Time critical commanding (say, to synchronize transitions within a multi-stage servo) is done within RTC and without the intervention of a high-level command by using an IPC mechanism. Both techniques are described in the next section.

4. DISTRIBUTED ARCHITECTURE

Distributed real-time computing involves coordinating a (possibly large) number of separate computers to solve a single set of tasks. Systems may be designed for a distributed topology for a variety of reasons, including:

1. **Locality:** needing pieces located next to hardware which is itself distributed.
2. **Performance:** needing multiple computers to finish the task within time constraints.
3. **Compatibility:** needing to integrate components which are already hosted on systems which can not be augmented to host a new application.
4. **Expandability:** the application may be required to support additional components at some time in the future. If designed as a distributed system, this future expansion can be done without redesigning the entire system.
5. **Diversity:** in some cases this is the best solution for inherently heterogeneous systems, for integrating components written in different computing languages, or for combining applications written for different computing platforms.

Heterogeneous distributed systems must account for variations in machine architectures on all nodes of the system. Some processors store data in "little-endian" form, where the first byte stored at a specified address corresponds to the least significant bits of the value, and other processors may store data in "big-endian" form, where the first byte stored at a specific address corresponds to the highest bits of the value.

RTC is designed for use in heterogeneous distributed systems. Mechanisms are provided to allow commanding, light-weight inter-process communication (IPC), and telemetry collection and storage. Accommodations are made for matching conventions across computing platforms.

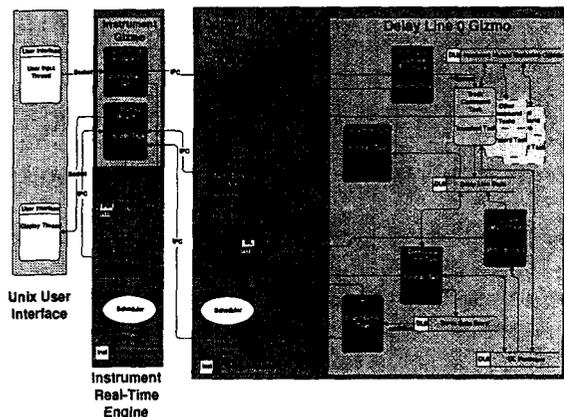


Figure 3. RICST Delay Line Object

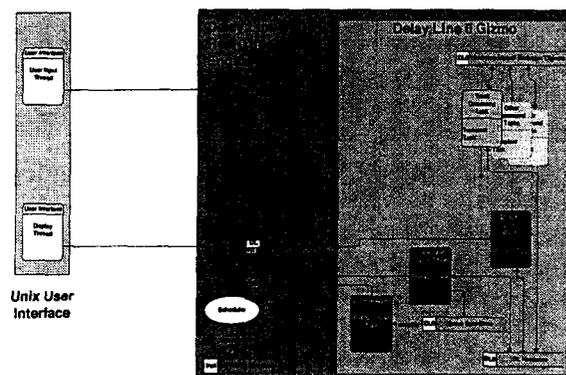


Figure 4. RTC Delay Line Object

Initial development of RICST had focused on the real-time requirements of systems, and used a custom socket-based approach for distributing binary packets of information for commanding and for telemetry. As the system matured, and the focus shifted to integrating this system with surrounding applications, it became increasingly difficult to manage

these interfaces and to support implementations for languages such as Java and Lisp. The adoption of CORBA for our communication framework neatly solved that problem. Transitioning RICST to become RTC was relatively straightforward, because the initial design of RICST modules mapped well to the capabilities and assumptions provided by CORBA. So, much of our code remained the same, while other code could be simply replaced by equivalent functionality provided by CORBA. In fact, we were able to eliminate several thousand lines of custom code from our system by moving to CORBA. Figure 3 is a high-level diagram of a delay line object in RICST which appeared in [3350], while Figure 4 shows the components appearing in RTC today for equivalent functionality; the remaining functionality is provided by the CORBA infrastructure.

1. IPC

Our IPC capabilities for doing double-buffered and first-in/first-out (FIFO) communications over local memory and shared memory have been described previously². These high-speed, localized implementations typically assume the presence of one writer and (possibly) multiple readers. We have since implemented IPC drivers to cover Ethernet, IEEE-1394 (Firewire), and IEEE-1596 interfaces, with minor changes to our IPC API to more transparently support these new components. These protocols can allow multiple writers to participate since they packetize data transfers in an atomic manner, but in practice most of our use cases involve point-to-point communication.

2. CORBA

CORBA defines a collection of standard interfaces and protocols to address distributed-processing issues. But the most obvious effect of having a CORBA-enabled application is that a server object (e.g. an optical delay line) appears to a client as though it were co-located with the client application.

We use several Object Request Brokers (ORBs) to implement a full RTC system. TAO^{**} is used for the real-time system, as well as for C++ servers or clients on Unix boxes, and has the nice benefit of using ACE^{***} which provides enhanced portability across platforms. JacORB provides our Java support, working around problems with the default CORBA support in Sun's Java. Mico is used to provide Tcl-based engineering controls and displays. omniORB is used for Python code, and ORBit is used for simple Perl clients.

CORBA includes standards covering several areas, which taken as a whole provide the features necessary to do distributed computing. (It is only recently, starting with the CORBA 2.2 specification, that the standard includes enough features to make good on this claim.) The standard includes:

1. An Interface Definition Language (IDL) which can be used to generate stub or glue code for specific programming languages. An IDL compiler is typically provided as part of the ORB package.
2. A generic communications protocol layer (GIOP; pronounced "gee-op") and a specific protocol for TCP/IP (IIOP; pronounced "eye-op").
3. Language mappings for C++, C, Python, and a few other languages. Other languages like Tcl and Lisp have mappings which are vendor-specific, but which may be proposed as an approved standard mapping in the future.
4. Standard services for common operations, including a "Naming Service" to enable discovery of available application servers.

RTC defines several interfaces in CORBA:

1. A simple "ping" interface to test for health of a remote object.
2. A core interface to define structures and some data types, as well as a processor management interfaces.
3. A configuration interface to allow reading and writing configuration values.
4. A hardware interface to expose boards and input/output channels associates with those boards.
5. A telemetry interface to send and receive telemetry. Built on top of CORBA Event Channels.
6. A system identification interface to allow characterization of components.

^{**} The ACE Orb was developed by a team lead by Douglas Schmidt. <http://www.cs.wustl.edu/~schmidt/TAO.html>

^{***} Advanced Communication Environment, Douglas Schmidt, <http://www.cs.wustl.edu/~schmidt/ACE.html>

Additionally, several basic interfaces for interferometry-specific devices such as delay lines and cameras are included to support commonality between our various implementation projects.

Refer to [4848-32] for a discussion of simple CORBA clients in C++, Python, Tcl, and Perl.

5. CONFIGURATION ARCHITECTURE

Virtually every real-time component in an RTC system is configurable. Not only does this allow for trivial remapping of hardware boards in a system, but also enables full and complete configurability for which kinds of software objects are to be used in a system, how many of each will be used and how each will behave. These general capabilities include configurable modules and strategies to provide:

1. Processor initialization.
2. Dynamic module loading.
3. Dynamic object configuration.
4. Dynamic object reconfiguration.

Configurable objects typically are derived from the `tRTCObject` class, which gives the module the notion of a name and a parent and the notion of a configuration manager. The name and parent information are used to map instantiated objects into a configuration heirarchy provided by an external configuration server. `tRTCObject` provides a large portion of the support code required to be a configurable object. Much of this functionality is usable without change, which allows derived objects such as, say, a delay line, to be configurable without much additional code.

A configurable object will implement `Configure` and `ConfigureSelf` methods. This allows the object to be externally commanded to dynamically reconfigure itself and all of its children, or just itself.

Every configurable object can respond to a reconfiguration request. It is the responsibility of the configurable object to ensure that all child objects are also reconfigured at that time. A configurable item is set by calling the method `DatabaseConfiguration`, which is overloaded to support the various data types to be configured. The common data types (e.g. integer and double) have predefined implementations for `DatabaseConfiguration`, while a new custom type may need an overloading function defined.

High-level objects (gizmos) can be dynamically assigned to processors. If applications are built using RTC features such as IPC, an object can be reassigned to another processor or moved to a new processor by simply changing configuration information and rebooting the system. In one example at JPL, a testbed team decided to change their system from one processor to two to split the processing load, and was up and running in the two-processor configuration within an hour or two from their decision to make the change. This included remapping objects which required IPC communication between themselves, which were commanded by remote clients, and which provided telemetry to these clients and to a data archive. But most of the elapsed time was spent configuring hardware and in understanding which configuration parameters needed to change (most of the team has not yet run a multi-processor configuration), and little time was spent actually reconfiguring software.

RTC draws virtually all of its configuration information from an external source. We refer to this as a configuration database, though it is not necessarily required to be a database in the common understanding of, say, a relational database such as Sybase or PostgreSQL. We have a complete implementation of a full-featured configuration management system which supports named sets of parameters, versioned sets of parameters, and parameter linkage between related items. This system has two components known as `ConfigServer` and `Configurator`, where the former provides configuration information to RTC and the latter provides a user interface to manage the configuration information. `ConfigServer` uses an RDBMS to store configuration information, allowing transaction integrity for data changes.

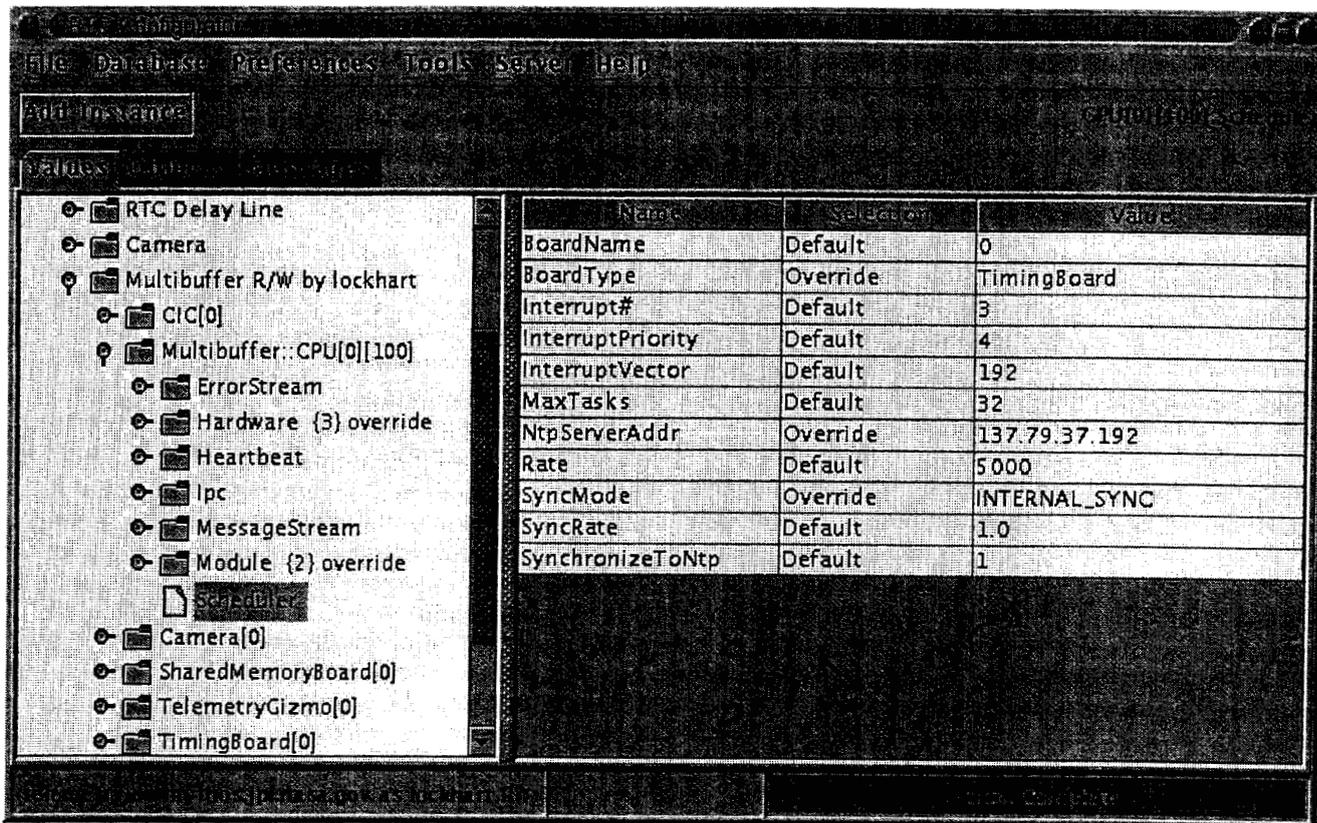


Figure 5. RTC Configuration user interface

We have demonstrated other configuration server implementations, including one in just one hundred lines of Python code using simple keyword/value pairs stored in a flat file (which itself was generated by a data dump from Configurator). This test implementation demonstrated repackaging possibilities for supporting RTC in a flight environment such as Space Interferometry Mission (SIM) [4852-01].

6. USER INTERFACES

We have developed a full-featured graphical user interface (GUI) for use with RTC. The GUI is written in Java and interfaces to the real-time system through CORBA-based command and telemetry interfaces. It has the ability to display telemetry in both numeric and graphical form, and sets of displays can be configured to come up automatically when the application starts.

All aspects of a running RTC-based system can be viewed through this GUI. Typically, a subset of the total available information will be displayed at any one time. RTC allows multiple clients to connect to telemetry and to connect to objects for commanding, so it is possible to run more than one instance of the GUI on different machines. We have run GUIs at JPL in California connected to the Keck Interferometer in Hawaii to allow our engineers to work with folks on site. An example GUI display is in Figure 6, showing a command window for a delay line, along with several telemetry display windows.

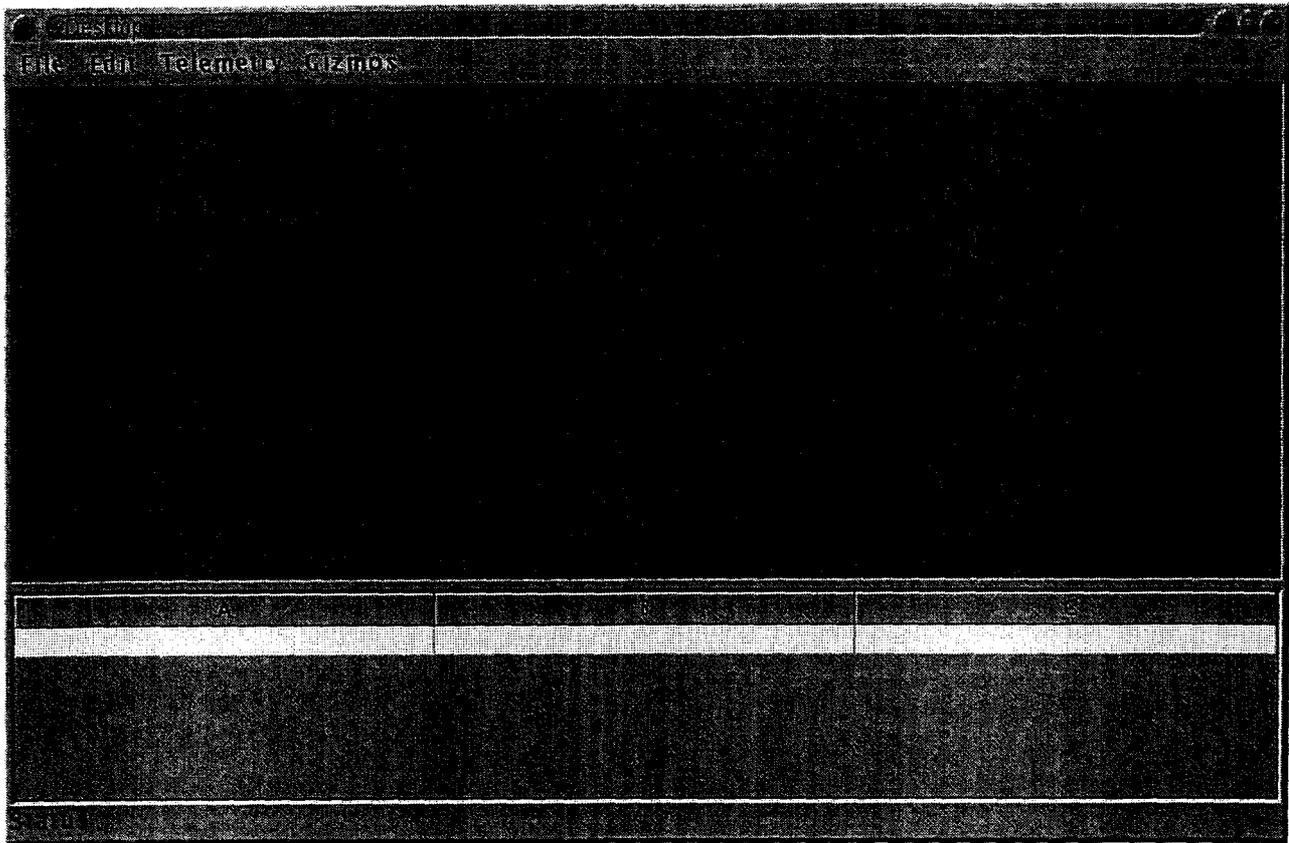


Figure 6. RTC graphical user interface

Real-time implementations such as the Keck Interferometer ~~(KI)~~ [4838-10] produce large amounts of telemetry (well above 1MB/sec). Handling this volume of data within a Java application has taken some special care to avoid performance issues provoked by object-proliferation and subsequent garbage collection.

7. CHALLENGES WITH NEW TECHNOLOGIES

There was a happy coincidence of requirements from RTC and new capabilities in the CORBA standard which allowed us to make a transition to CORBA. The transition would not have been possible earlier because the CORBA standard and individual CORBA implementations were up to the task.

Adopting COTS technology (and, in fact, using any software developed outside an organization) trades the effort and time required for implementing code for the effort and time required to understand the “imported” package. The payback for adopting industry-standard technologies is not always immediately quantifiable.

Open-source COTS (or otherwise OTS) software can help hedge against the risk of choosing a particular technology or implementation, particularly for projects with long implementation cycles or long operational lifetimes.

8. FUTURE DIRECTIONS

As our real-time components become more mature, we will be pushing some capabilities such as our configuration interface into more of the non-real-time portions of our software. For example, several of our systems have or will have external, non-real-time high-level sequencers to assist with system operation and these sequencers can and should be configured using these techniques.

We are currently exploring the use of Python for scripting applications in our systems; it is a well-structured, OO language with good CORBA support and a wide variety of support libraries.

RTC has been ported to RTAI, a variant of a real-time Linux [4848-69], and we expect to have opportunities to work with other platforms.

9. SUMMARY

RTC has evolved substantially since the first implementations of RICST. We have adopted CORBA as the middleware infrastructure for distributed computing in JPL testbeds and at the Keck Interferometer, and have developed a powerful configuration capability for RTC-based systems. This toolkit, though targeted for optical interferometry systems, is suitable for a wide range of applications in the real-time control arena.

ACKNOWLEDGEMENTS

The work performed here was conducted at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Braden Hines led the team developing much of the current capabilities of this system. JPL's Interferometry Real-time Systems and Software Group has several members who have made fundamental contributions to RTC and its applications: Richard Johnson is the RTC Software Architect, providing ongoing technical leadership for this work; Phillip Irwin and Elizabeth McKenney have made many important contributions over the life of the project and have deployed RTC into several new interferometer testbeds; Michael Deck of Cleansoft Inc. developed much of the configuration user interface; Glenn Eychaner has developed our Java-based user interface; and Courtney Duncan, Graham Hardy, Erik Hovland, Al Niessner, and Martin Regehr are building testbeds with RTC and contributing to the next generation of software.

REFERENCES

1. B. E. Hines, R. L. Johnson, K. M. Starr, "Common interferometer control systems architecture", Proceedings of SPIE, Volume 3350, pp. 644-653.
2. R. L. Johnson, Jr., E. A. McKenney, K. M. Starr, "Real-time control software for optical interferometers: the RICST testbed", Proceedings of SPIE, Volume 3350, pp. 153-162.
3. P. C. Irwin, R. Goullioud, "Hardware design and object-oriented hardware driver design for the Real-time Interferometer Control System Testbed", Proceedings of SPIE, Volume 3350, pp. 146-152.